373 Prerequisite Knowledge Refresher

This is an optional resource to help you review all the prerequisite knowledge you need going into CSE 373 this summer. It was generously created by 373 veterans Zach Chun and Yuma Tou. You can email them if you spot any typos or have suggestions for improving this document!

Table of Contents

 # Definitions 2
Abstract data types (ADTs) 2
Data structures 2
List ADT 3
Map ADT (Dictionary ADT) 3
 # Data structures review 3
Main takeaways 3
Arrays 4
Practice-it problems 5
Node objects 5
Practice-it problems 7
ArrayList and LinkedList (AL and LL) 8
AL and LL: summary 8
AL and LL: full explanation 8
Practice-It problems 13
Doubly-linked list 13
Binary trees 14
Practice-it problems 17
 # Java review 17
Methods 17
Classes and objects 18
Reference semantics 20
Interfaces 23
Data structure implementations in Java 24
Generics 24
Wrapper classes and autoboxing 25

Definitions

Note: These will be the standard definitions we use in class. Our definitions may differ slightly from information found on the internet because there are slight differences in opinions from different sources. Despite these slight differences, the general ideas are the same.

Abstract data types (ADTs)

An abstract data type (ADT) is a theoretical model (not a concrete programming idea) that defines expected operations and behavior without specifying the specific implementations for the behavior.

For example, a "list" is a common ADT in programming. A list is a collection of ordered elements. The expected behavior for a list includes accessing elements by index, adding or removing elements at any index, getting the total number of elements, etc. The List ADT only defines the expected behavior for something to be considered a list, but does not and should not define the specifics of the implementation, such as how elements would be stored or how this behavior would be implemented in code. When we write pseudocode or a general description of an algorithm, we often reference ADTs to focus on the ideas of how the algorithm works as opposed to describing specific implementation details that do not matter in the context of the pseudocode.

Data structures

A data structure is a way of storing data and performing operations on this data. It is a concrete implementation, unlike an abstract data type (ADT) which is an abstract definition. However, data structures can fulfill an ADT, meaning that it provides all of the required behaviors of that ADT. Examples of data structures include arrays, trees, and linked lists (node objects). Out of these, arrays and linked lists are data structures that can be used to satisfy the list ADT.

Many programming languages, including Java, provide data structure implementations for us to use in our programs, such as ArrayList, TreeSet, and HashMap. These implement the list, set, and dictionary/map ADTs respectively. (These ADTs and data structure implementations will appear later in the course.)

List ADT

A List is a common abstract data type (ADT) in programming. A list is a collection of ordered elements. The expected behavior for a list includes accessing elements by index, adding or removing elements at any index, getting the total number of elements, etc. The number of elements in a list is not fixed; a list can grow or shrink in size according to the number of elements stored.

Map ADT (Dictionary ADT)

A Map is another common abstract data type -- a map (also called dictionary or associative array) is a collection of pairs: each pair has a key and an associated value. The most common operation is to look up what the value is for a given key -- this requires that all the keys are all unique. The number of elements in a map is usually defined as the number of key-value pairs. Like the List ADT and most other ADTs, the number of elements is not fixed and can grow or shrink in size as pairs are removed or added.

See recent CSE 143 slides for uses of maps and visuals, as well as example code using Maps in Java:
https://courses.cs.washington.edu/courses/cse143/19au/lectures/10-16/slides/maps.pdf

Data structures review

Main takeaways

- An array is a data structure that can hold a fixed number of elements. Any index can be accessed or modified in constant runtime.
- Node objects are objects that have fields to keep track of other nodes to represent a collection of data. They're most often used to represent a linked list, but can be more flexible than that because we control the design of the node class.
- ArrayList and LinkedList are two possible implementations of the List ADT/interface, and they use arrays and nodes respectively to organize data in a linear fashion.
- There are trade-offs between ArrayList and LinkedList, and when implementing using a List in our code, we should try to evaluate the situation and how we'll be

using our list (are we inserting or deleting from the front frequently? `get`ing from the middle?) to make a good design decision.
- Nodes and arrays are the core building blocks of many data structures, and we'll see them reused in similar and new ways to implement much more than just the List ADT.

Arrays

An array is a data structure that holds a fixed number of items of a certain type. The type of items it can hold is specified when it is declared and the size of the array is specified when it is constructed. Each of the spots of an array is referred to by its index. The first element in an array is at index 0, the next is at index 1, etc., and any element stored in an array can be directly accessed or modified by using its index.

The following code will construct a new array:

```
int[] numbers = new int[8];
// creates an array with 8 spots for integers [0, 0, 0, 0, 0, 0, 0, 0]
```

When arrays are constructed as above, they are filled with the default value for the type. The default value for int (and other numerical types such as double) is 0, so this array will start off storing 8 zeros. The default value for all object types is null.

There is an alternate way to construct an array if it is already known what values the array should hold at the time of construction:

```
char[] letters = {'h', 'e', 'l', 'l', 'o'};
// creates an array with 5 spots, storing the 5 provided chars
```

Regardless of which method is used to construct an array, its capacity cannot be changed after construction. The first array we constructed (`numbers`) will always have a total size of 8, while the second array we constructed (`letters`) will always have a total size of 5.

Arrays can be modified to store different values by using the indices. If we know the index of the value we want to update, we can do so directly.

```
numbers[0] = 5;  // sets value at index 0 to 5, the array now holds [5, 0, 0, 0, 0, 0, 0, 0]
numbers[2] = 8;  // sets value at index 2 to 8, the array now holds [5, 0, 8, 0, 0, 0, 0, 0]
```

Below is a code example that sets the value at all indices of the array to 28, so that the array stores only 28's.

```
for (int i = 0; i < numbers.length; i++) {
   numbers[i] = 28;
}
// now numbers stores [28, 28, 28, 28, 28, 28, 28, 28]
```

For arrays of objects, the idea of reference semantics still applies here, meaning that an array of objects holds a fixed number of references to objects of the specified type.

Note: The `Arrays.toString(array)` method can be used to convert an array into its String representation. This can be especially useful for debugging purposes.

Practice-it problems

- arrayMystery5
  https://practiceit.cs.washington.edu/problem/view/3900?categoryid=120
- findMin
  https://practiceit.cs.washington.edu/problem/view/cs1/sections/section7/findMin
- sum5
  https://practiceit.cs.washington.edu/problem/view/cs1/sections/section7/sum5

Node objects

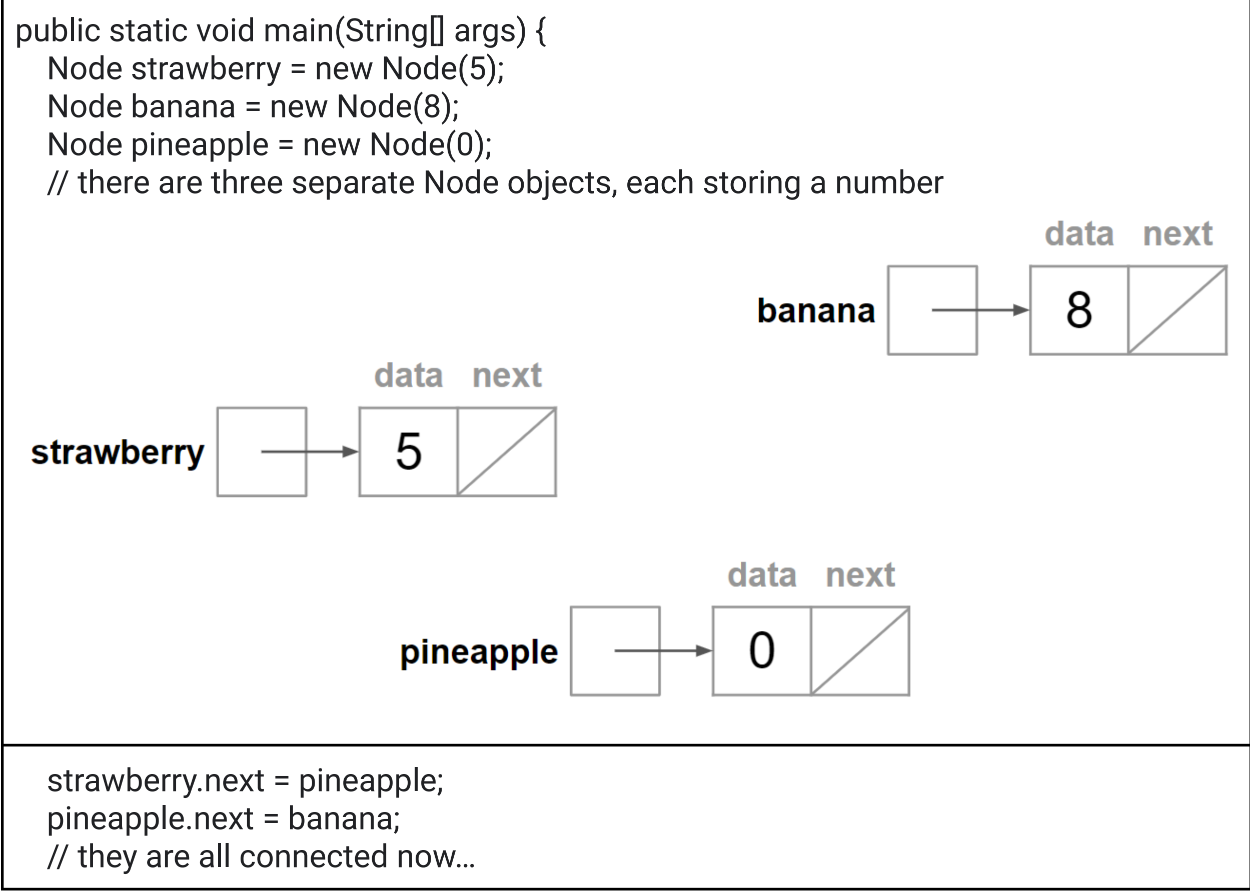
Recall that we can define new classes and use them, so we can come up with our own ways of storing data outside of what has been provided already by Java (such as arrays). We could define a Node class that acts as a blueprint for Node objects. These Node objects could each store a piece of data, and we can keep track of multiple pieces of data by having these Node objects reference each other.

This Node class is defined below. Each Node stores a piece of data (in the data field) and a reference to another node (in the next field). For simplicity, our Node class will only be able to store integer values.

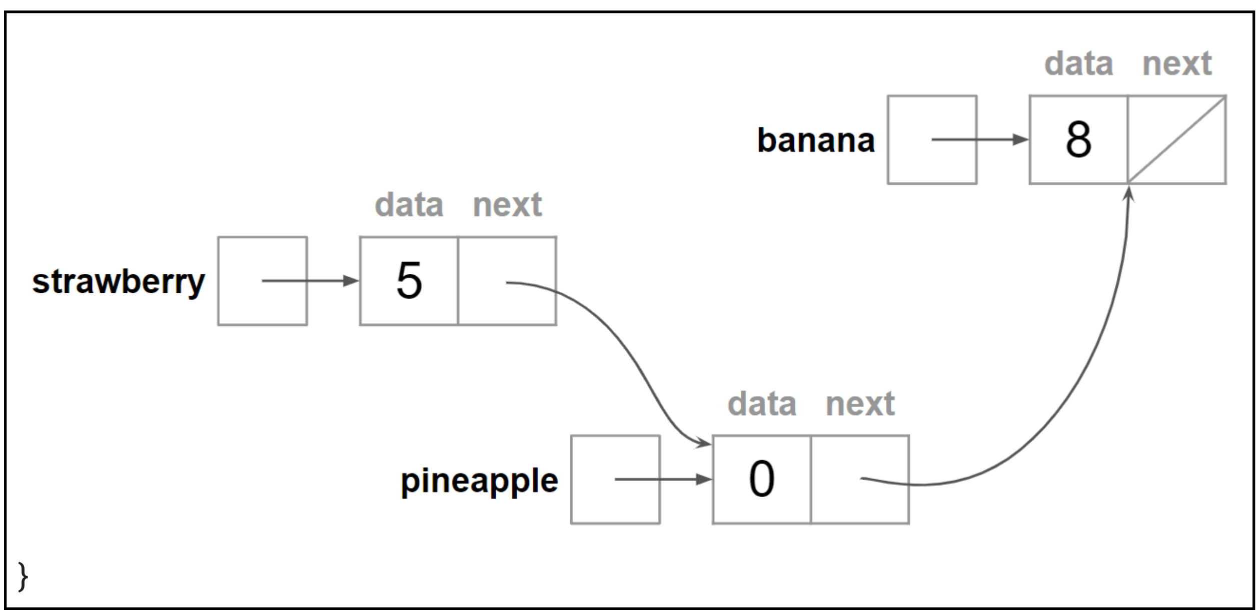
```
public class Node {
    public int data;
    public Node next;

    public Node(int data) {
        this(data, next);
    }

    public Node(int data, Node next) {
        this.data = data;
        this.next = next;
    }
}
```

Example code showing how we might store a collection of numbers (the numbers 5, 0, and 8) using these Node objects:

```
public static void main(String[] args) {
    Node strawberry = new Node(5);
    Node banana = new Node(8);
    Node pineapple = new Node(0);
    // there are three separate Node objects, each storing a number
```




```
    strawberry.next = pineapple;
    pineapple.next = banana;
    // they are all connected now…
```



We can see that each Node object stores one of the numbers in the `data` field, and that all three of the Node objects are connected to each other by references stored in the `next` field. If we keep track of the `strawberry` variable, we can access all three of the numbers being stored since we can reach the other Nodes from `strawberry`. By using Node objects that we defined ourselves, we are able to store a collection of numbers.

This illustrates the idea that we can define our own Java classes to store data without relying on pre-existing Java objects like arrays. This example situation only has 3 nodes and stores them in a linear way, but there might be situations where we might want to store large amounts of data, store data with more complicated relations, or provide more complex or very specific operations on the data stored. In these situations, the ability to define our own node class like this would be much more useful because we can customize it to best fit the use case.

Practice-it problems

- https://practiceit.cs.washington.edu/problem/view/cs2/sections/linkednodes/problem5
- https://practiceit.cs.washington.edu/problem/view/cs2/sections/linkednodes/problem9

ArrayList and LinkedList (AL and LL)

AL and LL: summary

- ArrayList stores its elements at the beginning indices of an array that generally has more capacity than the current number of elements in the list, so that adding new elements doesn't always require resizing. Because all the elements are at the front of the array, if something is inserted at the beginning of the array, all the existing elements must be shifted over to maintain the ordering. Accessing elements with the `get` method, however, is always very efficient.
- LinkedList stores a reference to the front node, which will store a reference to the second node, which will store a reference to the third node, and so on. This implementation is more efficient than ArrayList when inserting or deleting values at the beginning indices of the list, as no shifting is required when inserting or deleting index 1, for example. However, to access elements in the middle or at the end of the list, LinkedList has to loop over all the nodes that come before to reach those elements unlike ArrayList.

AL and LL: full explanation

ArrayList and LinkedList are two different implementations of the List interface in Java (and the List ADT). A List is a collection of ordered elements. In Java, the List interface requires that any object that implements List (i.e., claims to be a List) contain certain methods, including but not limited to:

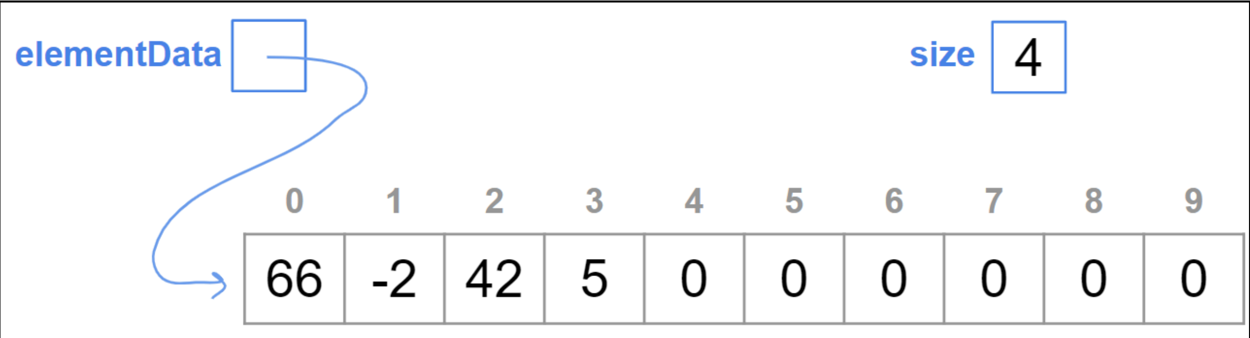
- `add(int index, E element)`
- `add(E element)`
- `get(int index)`
- `remove(int index)`
- `remove(E element)`
- `set(int index, E element)`
- `size()`

The ArrayList and LinkedList implementations of a List use different methods of storing elements, but both fulfill the List ADT by representing a collection or ordered elements and providing the required operations (add, remove, etc.).

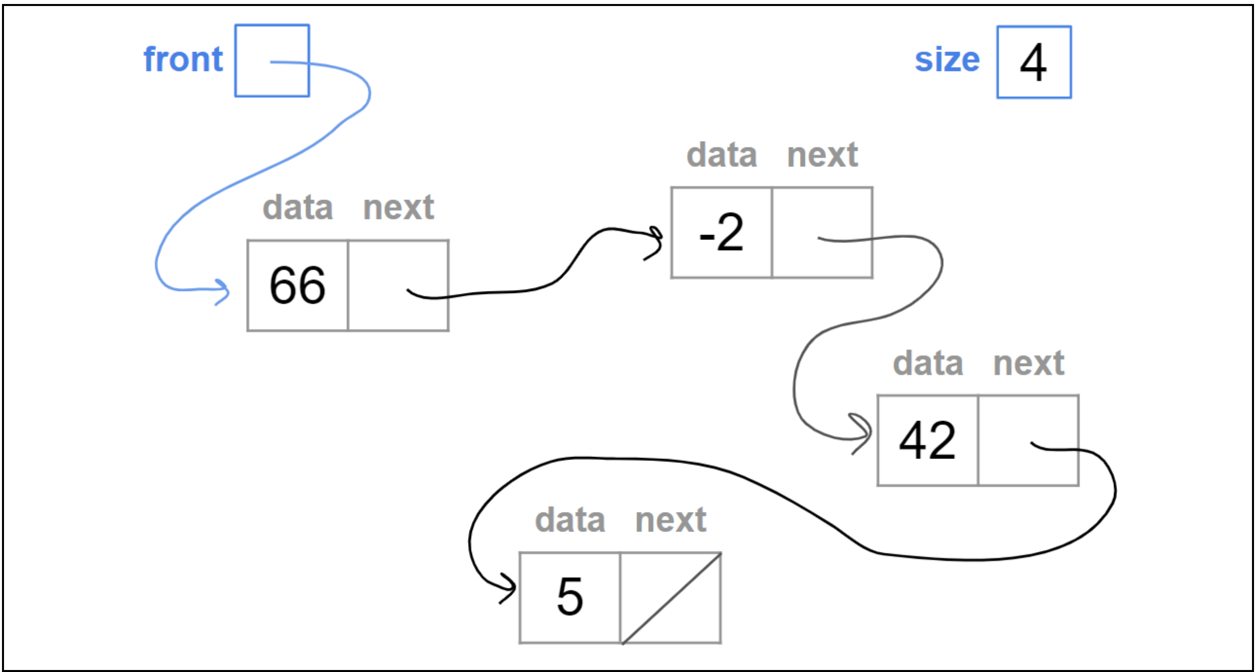
A List of ints that contains the numbers 66, -2, 42, and 5 (in that order) would store the numbers internally in different ways depending on whether the implementation is ArrayList or LinkedList. However, regardless of implementation, the List would represent

[66, -2, 42, 5] and provide the same functionality. Below shows how the internal representation would differ between ArrayList and LinkedList for the same abstract idea of a List of [66, -2, 42, 5].

ArrayList



LinkedList



The internals of ArrayList and LinkedList are quite different, but they both represent the same List.

An ArrayList uses an array as the storage structure for its elements. An ArrayList uses an array as the storage structure for its elements. Generally, the array is initially constructed with more capacity than the current number of elements so that the

insertion of a new element does not always force the creation of an entirely new bigger array.

Note that because of this, the size of the array is not necessarily equivalent to the size of the ArrayList (which would be the number of elements being stored in the ArrayList). This means that an ArrayList must also remember the number of elements that are currently stored in itself.

In the ArrayList image above, elementData is the array field and size is the field storing the number of elements currently stored in it.

A LinkedList uses linked node objects as storage for its elements. Each node object stores the element along with a reference to the node object storing the element at the subsequent index. The LinkedList itself stores a reference to the node object storing the first element (the element at index 0). This way, it is able to access all of the elements since it can repeatedly follow the references to the next node object until the last node is reached.

In the LInkedList image above, front is the field that stores a reference to the node object storing the first element in the list and size is the field storing the number of elements currently stored in it.

Below is example code showing the differences in implementation of the `toString` method (which generates a String representation of the list and its contents) between ArrayList and LinkedList.

```
public class ArrayList<E>
        implements List<E> {

    private E[] elementData;
    private int size;

    public String toString() {
        if (size == 0) {
            return "";
        } else {
            String result = elementData[0];
            for (int i = 1; i < size; i++) {
```

```
public class LinkedList<E>
        implements List<E> {

    // refers to first node (null if empty)
    private ListNode front;
    private int size;

    public String toString() {
        if (front == null) {
            return "";
        } else {
            String result = front.data;
```

```
            result += " " + elementData[i];
        }
        return result;
    }
  }
}
```

```
        ListNode current = front.next;
        while (current != null) {
            result += " " + current.data;
            current = current.next;
        }
        return result;
    }
  }

  private class ListNode {
      E data;  // data stored in this node
      ListNode next;  // link to next node in list

      public ListNode(E data) {
          this(data, null);
      }

      public ListNode(E data, ListNode next) {
          this.data = data;
          this.next = next;
      }
  }

}
```

Note how both implementations of `toString` have to loop over all of the elements inside their fields to be able to use them in the String result, but the syntax for doing so is different.

The next two methods we'll compare are the `get(int index)` which returns the value at that index of the list, and `add(int index, E value)`, which adds the given value at the given index in the list. Unlike toString, the internal behavior for these two methods will really showcase the differences between the implementations beyond just syntax.

```
public E get(int index) {
    return elementData[index];
}
```

```
public E get(int index) {
    ListNode current = front;
    for (int i = 0; i < index; i++) {
```

```
        current = current.next;
    }
    return current.data;
}
```

For `get(int index)`, ArrayList can take advantage of the fact that arrays can instantly jump to any index in the array and access or modify the data in that position. LinkedList, on the other hand, has to loop through all the elements until they get to the index specified by the parameter. We can imagine that ArrayList's implementation for get is faster than LinkedList's implementation in most situations. For example, if we call get and pass in the index at the back of the list, the ArrayList code will only have to execute one line of code for the whole method, while the LinkedList code will have to execute the `current = current.next` line for as many elements as there are in the list. To introduce some notation, the one on the right runs in O(n) runtime in the worst case, where n is the variable representing how many elements are in the list, since it has to potentially execute order-n lines of code. The one on the left runs in O(1) runtime in all cases, since it always executes 1 line of code.

```
public void add(int index, E value) {
    for (int i = size; i > index; i--) {
        elementData[i] = elementData[i - 1];
    }

    elementData[index] = value;
    size++;
}
```

```
public void add(int index, E value) {
    if (index == 0) {
        // insert at the front
        front = new ListNode(value, front);
    } else {
        // insert in middle/end; walk to node before the one to insert
        ListNode current = front;
        for (int i = 0; i < index; i++) {
            current = current.next;
        }
        current.next = new ListNode(value, current.next);
    }
}
```

For the `add(int index, E value)` method and these implementations, it's less clear which implementation is better. For example, if we insert at index 0 with an ArrayList, that for loop will run size times (i.e., it will have to loop over every element), since it has to shift over every value to the right by one in the array to make an empty space for the new value. For LinkedList on the other hand, add at the beginning of the list will not have to do any looping at all -- it can enter that first if branch and only execute a constant number of lines of code, which is independent of the size of the list. It turns out that

add'ing (index 1 or 2) near the front is efficient in a LinkedList since we only will have to loop the specified index number of times (which is small if the index is at the front).

If we add at the very end of this list, however, LinkedList will have to loop through every element to reach the current end node that we want to attach on the new ListNode to. This is in contrast to ArrayList, where none of the elements will have to be shifted over to the right (see how the for loop runs however many times `size - index` is).

Practice-It problems

- ArrayIntList (ArrayList specifically for integers)
    - isConsecutive https://practiceit.cs.washington.edu/problem/view/cs2/exams/midterms/midterm17/isConsecutive
    - insertAt https://practiceit.cs.washington.edu/problem/view/cs2/exams/midterms/midterm25/insertAt
- LinkedIntList (LinkedList specifically for integers)
    - min https://practiceit.cs.washington.edu/problem/view/bjp4/chapter16/e2-min
    - deleteBack https://practiceit.cs.washington.edu/problem/view/bjp4/chapter16/e7-deleteBack

Doubly-linked list

There are some standard optimizations we can make to our LinkedList to improve the runtimes of its methods in some cases:

- Keep track of the last node in the list so we can easily access the back, where we would previously have to loop all the way from the front to get to it.
- Have each node keep track of the previous node in addition to the next node, so we can traverse from back to front if appropriate. Say we wanted to access the 2nd-to-last element or the 3rd-to-last element. We should loop from the back to the front so we don't have to execute as many loop iterations as if we looped from front to back.

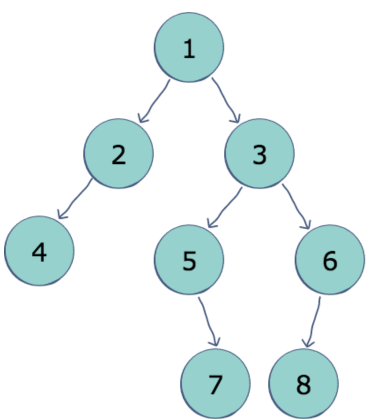
Take a look at the following improved-runtime DoubleLinkedList class:

```
public class DoubleLinkedList<E> implements List<E> {
    private ListNode front;
    private ListNode back;

    // methods...

    private class ListNode {
        E data;
        ListNode prev;
        ListNode next;
    }
}
```

Although this is faster for accessing or modifying towards the back of the list, there is some slight memory cost, as there are twice as many references stored. Depending on the current situation and requirements, there may be some situations where memory is limited or we know we're only frequently working with elements in the front of the list, so the singly linked list is better. Or maybe we know that we are working with both the front and back, and that we care about our program being as fast as possible. In such a case, the doubly-linked list is more appropriate.

Binary trees

Binary trees are a data structure that's made up of node objects where each node can have a maximum of 2 other nodes that it points to (called children). Just like with linked lists, there is a start point, called the root, that can reach all the children further down in the tree by following the pointer fields.



In the picture to the left, the node with 1 is the root of the tree. It can access its two children, the 2 and the 3 nodes, directly via its fields. Note that the 2 node in the second level of the tree has exactly 1 child, but this still satisfies the definition of the binary tree where each node has a maximum of 2 children. The nodes with 4, 7, and 8 data at the bottom of the tree are called leaf nodes because they have no children.

Just like arrays and nodes, binary trees are a fundamental data structure idea that can be used to implement several ADTs. In CSE 143, we practiced manipulating binary trees in code, but in CSE 373 we will later discuss how binary trees can be used to implement various useful ADTs like Set, Map, and PriorityQueue.

Below is part of an example implementation of a binary tree, showing its field (`overallRoot`, a reference to the topmost node of the tree) and the node class (`TreeNode`) which has fields to store the data and up to 2 references to children nodes.

```
public class Tree<E> {
    private TreeNode overallRoot;

    public Tree() {
        overallRoot = null;
    }

    // methods here

    private class TreeNode {
        public E data; // data stored at this node
        public TreeNode left; // reference to left subtree
        public TreeNode right; // reference to right subtree

        // Constructs a leaf node with the given data.
        public TreeNode(int data) {
            this(data, null, null);
        }

        // Constructs a leaf or branch node with the given data and links.
        public TreeNode(E data, TreeNode left, TreeNode right) {
            this.data = data;
            this.left = left;
            this.right = right;
        }
    }
}
```

We can see that there is a node class that's similar to the node class used for linked lists, except that it contains two fields to other node objects instead of just one.

Below is the implementation for a method called `size()` that will return the total number of values stored in the binary tree. The idea represented in English is: If the currentRoot is empty, add 0 to the total size computation. Otherwise, add 1 to the size and keep traversing the tree to add the size of the left subtree summed with the size of the right subtree.

- Note that it uses recursion to traverse the binary tree, since the structure of the binary tree is irregular -- using iteration and loops here would require some more if/else cases so we prefer recursion.
- Additionally, note that the presence of a private helper method is required so that recursion can be used -- if the original method is used, there is no parameter to keep track of which subtree we are currently looking at, which is used to determine the base case or to recurse on (and reduce to the base case condition).

```
public int size() {
    return size(overallRoot);
}

private int size(TreeNode currentRoot) {
    if (root == null) {
        return 0;
    } else {
        return 1 + size(currentRoot.left) + size(currentRoot.right);
    }
}
```

Below is the implementation for a method called `addLeftmost` that will take in a value and make a new node containing that value in the leftmost position of the binary tree. The idea in English for this one is: traverse left down on the tree until we reach the end (represented by null). When we reach null, create the new node and return that new node to whoever called us so that the caller can store that result in the appropriate field. This method is similar to the one above in that it uses recursion and a private helper method to accomplish its behavior. This pattern is called `x = change(x)` in CSE 143 where we pass in information (x) to a method, and use the returned node reference to update x itself. This commonly comes up when we need to modify a binary tree recursively -- we'll often see that the return type of the private method is a tree node and that we want to update the left, right, and overallRoot with the result of calling the method.

```
public void addLeftmost(E value) {
   overallRoot = addLeftmost(value, overallRoot);
}

private TreeNode addLeftmost(E value, TreeNode currentRoot) {
   if (currentRoot == null) {
      return new TreeNode(value);
   } else {
      currentRoot.left = addLeftmost(value, currentRoot.left);
      return currentRoot;
   }
}
```

Practice-it problems

- binaryTreeProperties
  https://practiceit.cs.washington.edu/problem/view/bjp5/chapter17/s3-binaryTreeProperties
- countLeaves
  https://practiceit.cs.washington.edu/problem/view/cs2/sections/binarytrees/countLeaves
- completeToLevel
  https://practiceit.cs.washington.edu/problem/view/bjp4/chapter17/e14-completeToLevel

Java review

Methods

A method consists of one or more Java statements that are grouped together, often to accomplish a certain task. Methods can be created to be able to reuse code (and reduce redundancy), improve readability, and enable abstraction. Methods can take input as parameters and produce output as a return value and/or as a side effect (like printing).

Method headers have the following format:

```
public double methodName(int parameterName) {
```

```
    // method body (the contents of the method)
}
```

The components of a method header, from left to right:

- **Access modifier (optional)**: *public*, *protected*, *private*, or not specified (known as the default access level)
- **Return type**: the type of the return value, or *void* if the method does not return a value
- **Method name**: a name for the method, lowercase with subsequent words camelcased
- **An opening parenthesis**
- **Parameter(s)**: 0 or more parameters separated by commas, where each parameter has the type followed by a name for the parameter (the name is how the parameter will be referenced within the method body)
- **A closing parenthesis**

After the method header, the contents of the method (the method body) is placed between curly braces.

The main method is a special method that is executed when a Java program is run. The main method must have a method header of public static void main(String[] args).

Classes and objects

In Java, classes are the fundamental components where code is defined -- everything in Java is written inside of a class. Generally, there are two types of classes: classes that can be executed (with a main method) and classes that define a blueprint for something that can store data and perform specific behavior. Java has predefined some blueprint classes already (like String or ArrayList), but we can define new blueprint classes and use them. In this course, we will be focusing on the blueprint type of classes when we design and implement blueprints for data structures.

A class can define state and behavior. For blueprint classes, the fields (variables that are maintained throughout an object's existence) define how the state will be stored and the methods define the behavior.

As an example, we can write a Java class as a blueprint for a Dog. We will say that the state of a Dog is its name and breed, and that the behavior of a Dog is barking, licking, and getting older.

```
public class Dog {  // class header that gives the name of the class (capitalized)

    /*** fields: define how state will be stored ***/
    private String name;
    private String breed;
    private int age;

    /*** constructor ***/
    public Dog(String name, String breed, int age) {
        this.name = name;
        this.breed = breed;
        this.age = age;
    }

    /*** methods: define behavior ***/
    public void bark() {
        System.out.println("woof");
    }

    public void lick() {
        System.out.println(":)");
    }

    public void getOlder() {
        this.age++;
    }

    public String toString() {
        return "U･ᴥ･U  [ " + this.name + " (" + this.age + ") ]";
    }
}
```

In the example above, there are fields that store the state (name and breed) and methods that provide the behavior (barking, licking, and getting older). There is also a constructor, which is like a special method that is called once per instance of a Dog object. Calling the Dog constructor will construct, or create a new instance of, a Dog with its own copies of the state and behavior defined in the Dog class.

Objects are these instances of the blueprint class that follow all the definitions in the class. When we want to use a Dog, we can create a Dog object using the `new` keyword and a defined constructor. This will create a Dog object that will have the state and behavior defined in the blueprint for Dog. Multiple Dog objects can be constructed, and each of these will be its own instance of a Dog, meaning it has its own copies of the state and behavior originally defined in the Dog class. In the example code below, dog1 and dog2 are two separate Dog objects constructed using our Dog class definition as a blueprint. Changing the age of dog1 will not affect the age of dog2, and vice versa.

```
Dog dog1 = new Dog("Duke", "dachshund", 8);
Dog dog2 = new Dog("Schatzi", "German shepherd", 5);
dog1.getOlder();  // "Duke", "dachshund", 9
```

Note: Classes are different from Java's built-in primitive types, as primitives don't have any methods defined and can only represent one simple value. Additionally, all of the predefined Java classes can be identified by a capitalized name. For example, String is an object, but int is a primitive type and is not an object.

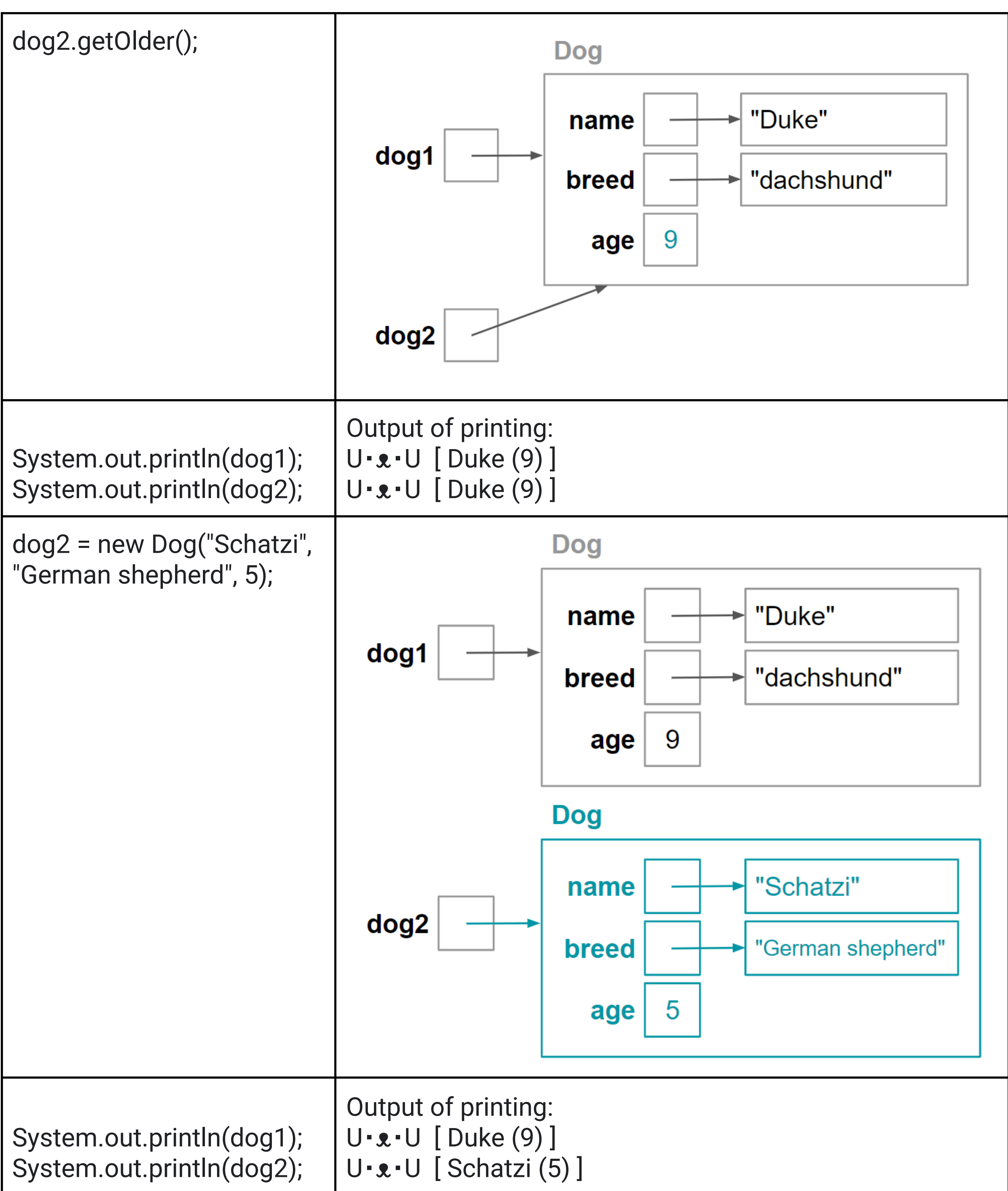
Reference semantics

Java uses reference semantics for all objects. This means that all object variables will store a reference to that object (we can think of it as storing the memory address where that object is stored), rather than storing the object itself, a copy of the object, or the object's value.

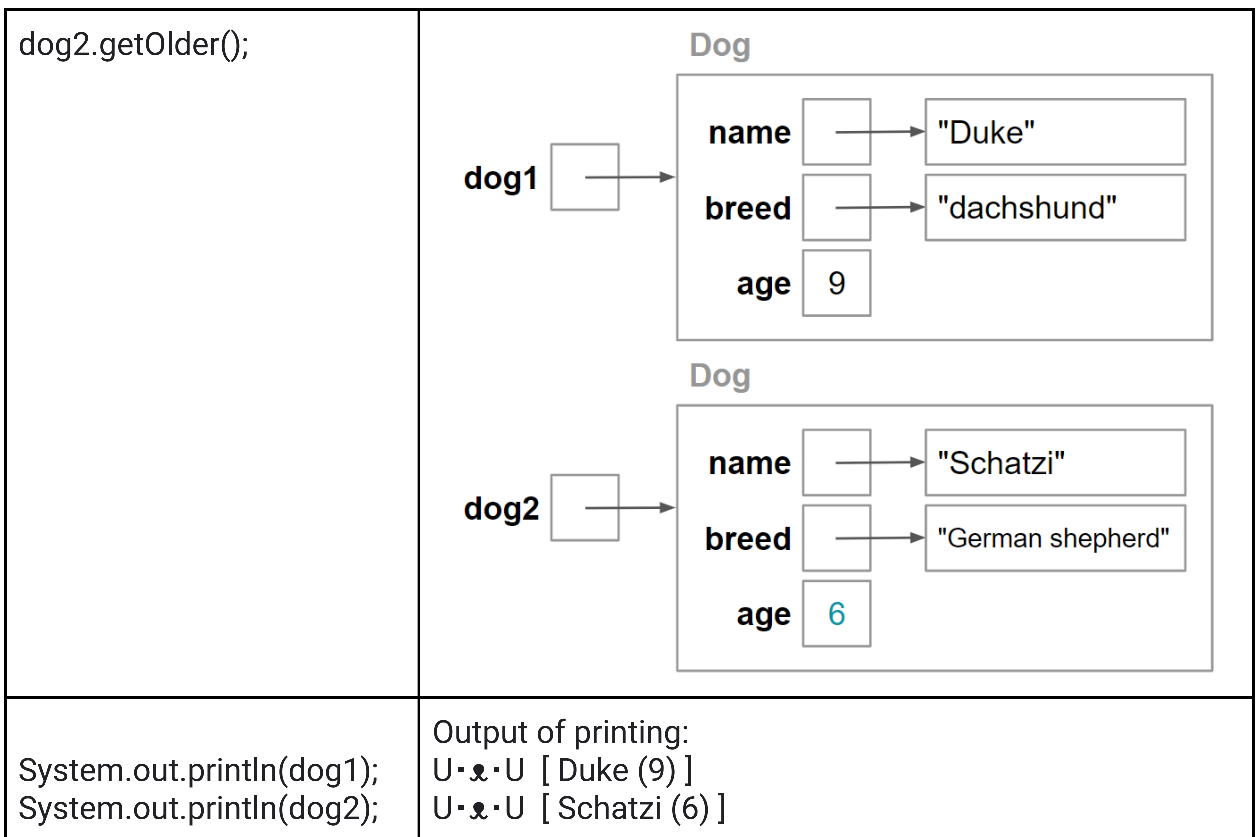
Java uses value semantics for all non-object, or primitive, types such as int, double, boolean, and char. This means that primitive variables will directly store the value of the primitive.

We will examine reference semantics in the following code examples and diagrams, using the Dog class given as an example in the **Classes and objects** section.

Dog dog1 = null; Dog dog2 = null;	dog1 dog2
dog1 = new Dog("Duke", "dachshund", 8);	Dog dog1 name "Duke" breed "dachshund" age 8 dog2
dog2 = dog1;	Dog dog1 name "Duke" breed "dachshund" age 8 dog2
System.out.println(dog1); System.out.println(dog2);	Output of printing: U・ᴥ・U [Duke (8)] U・ᴥ・U [Duke (8)]



dog2.getOlder();	
System.out.println(dog1); System.out.println(dog2);	Output of printing: U･ᴥ･U [Duke (9)] U･ᴥ･U [Duke (9)]
dog2 = new Dog("Schatzi", "German shepherd", 5);	
System.out.println(dog1); System.out.println(dog2);	Output of printing: U･ᴥ･U [Duke (9)] U･ᴥ･U [Schatzi (5)]



Interfaces

In Java, an interface is a definition for a collection of methods (name, parameters(s), and return type) without including any implementation. Essentially, it is Java's representation of an abstract data type (ADT).

For example, Java's List interface (which includes methods such as get, add, remove, and size) represents the List ADT.

If a class implements a certain interface, it must provide implementations for all of the methods defined by the interface. By implementing an interface, the class has an "is-a" relationship with the interface. In other words, the class promises that it can act as that interface when it is used. A single class can implement more than one interface.

Example:

```
public interface Shape {
    double area();
```

```
    double perimeter();
}

public class Triangle implements Shape {
    // fields…

    public double area() {
        // code to calculate and return the area of this triangle
    }

    public double perimeter() {
        // code to calculate and return the perimeter of this triangle
    }
}
```

In this example, Triangle implements Shape as it includes “implements Shape” in the class header and provides implementations for all of the methods defined by the Shape interface (`area()` and `perimeter()`). We can say that a Triangle “is a” Shape and that Triangle can be used as a Shape.

Data structure implementations in Java

Java provides some data structure implementations, such as the ArrayList, TreeSet, and HashMap classes. These implement the list, set, and dictionary/map ADTs respectively. (These data structures and ADTs will appear later in the course.)

Generics

Java’s provided core data structure implementations (such as ArrayList, TreeSet, HashMap, etc.) can store objects of any type because they are generic -- i.e., their class definitions specify that they can store elements of any specified data type.

As a client wanting to put Strings in an ArrayList, we would construct an ArrayList<String>. The type of object being stored in the ArrayList goes between the angle brackets. This is called a type parameter.

As the implementer of a generic data structure class, the class definition must include some special syntax to declare a type name that will be referred to throughout the implementation of this class.

```
public class Box<T> {
   // T stands for Type
   private T t;

   public void set(T t) {
      this.t = t;
   }

   public T get() {
      return t;
   }
}
```

Technically, we could get away with replacing generic types with Java's Object type, but doing so disregards the safety advantages of Java's type system. See this stackoverflow post for some extra thoughts: https://stackoverflow.com/questions/5207115/java-generics-t-vs-object

Wrapper classes and autoboxing

When using generic classes like ArrayList, the type parameter must be an object type, but there are cases where we want to store primitive types (such as int, double, char, boolean, etc.) in ArrayLists rather than objects. In these cases, we can use the "wrapper" class for the primitive type we want. For example, if we want to have an ArrayList of char elements, we can construct an ArrayList<Character> object for that use case. Some other wrapper classes are listed below.

Primitive type	*Wrapper class*
boolean	Boolean
char	Character
double	Double
integer	Integer

Each of these wrapper classes is Java's way of reformatting the primitive data types into object types so they can be used anywhere that object types are required. Each of

the wrapper classes stores a field of the primitive type version so it can keep track of that one piece of data. For example, the Double wrapper class looks roughly like:

```
public class Double {
   private double value;

   // useful methods to get the value or do operations on doubles
}
```

Luckily for us, Java automatically converts between primitive types and their wrappers classes for us, through a process called autoboxing. This basically lets us ignore the difference between the two when we write code like the following:

```
ArrayList<Integer> list = new ArrayList<>();
int x = 5;
list.add(x);
list.add(3);
int y = list.get(1);
Integer x2 = list.get(0);
```

See Oracle's documentation for more:
https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html